Integrating Testing into the Alloy Model Development Workflow

Allison Sullivan
University of Texas at Arlington
Arlington, TX USA
allison.sullivan@uta.edu

***Abstract*—Software models help improve the reliability of software systems: models can convey requirements, and can analyze design and implementation properties. A key strength of Alloy, a commonly used modeling language, is the Alloy Analyzer toolset. The Analyzer is an automated analysis engine that searches for all valid instances, which are assignments to the sets of the model such that all executed formulas hold, up to a user-provided scope. Unfortunately, despite the Analyzer, writing correct models remains a difficult and error-prone task. To address this, a unit testing framework, AUnit, was created for Alloy. Since then, several traditional imperative testing practices, including mutation testing, fault localization and repair, have been established for Alloy models. Prior work has introduced the feasibility of these approaches and produced command line prototype tools. This paper highlights the effort to translate these research products into the Analyzer, the main model development tool for Alloy, to produce one consolidated integrated development environment that provides robust testing support.**

***Index Terms*—Alloy, SAT Solver, Software Testing**

I. Introduction

As software pervades our society and lives, and software failures become increasingly costly, there is a growing need to produce higher quality software at lower costs. To achieve this, software developers can utilize bounded verification techniques where key portions of the system's design are modeled in declarative logic and then automatically analyzed up to a user-provided scope [3, 6, 12, 17, 33, 35, 44]. In addition to catching subtle but often dangerous bugs that can arise in designs, a precise model of a system's design also enables: architects to guarantee changes are safe before modifying the implementation, stakeholders to remove ambiguity about the system being built, and developers to produce better self-diagnosing code [24]. However, leveraging software models to catch early design bugs introduces a "chicken and egg" problem: to gain the many benefits that come from having a software model, the model itself needs to be correct. Unfortunately, writing correct models is a difficult task, in part because reasoning about the interaction between multiple formulas is difficult to do manually.

Alloy is a popular declarative, first order modeling language [13] that has been used to validate software designs [4, 8, 18, 23, 41, 44], to test and debug code [9, 19], to repair program states [27, 43] and to provide security analysis of systems [3, 5, 34]. A key strength of Alloy is the ability to develop models in the Analyzer, an instance enumeration toolset powered by SAT solvers that lets users explore their models by producing a collection of satisfying instances, which are assignments to the sets and relations of the model such that all executed formulas hold. At a conceptual level, each instance depicts behavior currently allowed by the modeled system. The SAT solver will then explore all possible behavior, potentially revealing unintended restrictions (or lack thereof) of the modeled system.

We created AUnit to address the need to have a systematic method to check the correctness of Alloy models [32]. Prior to AUnit, there was no formal notion of "testing" in the Analyzer. As a result, experienced users would employ a range of ad-hoc techniques, such as enumerating instances and visually inspecting them for issues, that are time consuming and error prone. AUnit's key insight is that unit testing, the most effective way to validate *code*, provides a blueprint on how to validate *models*. In the context of Alloy's declarative execution, in which there is no notion of imperative control flow and the SAT solver finds all satisfying scenarios in one execution, AUnit defines: (1) what is a test case, (2) how is a test case executed and its pass/fail outcome resolved and (3) what are different types of coverage criteria.

To start providing native support for testing in the Analyzer, we have previously extended the Analyzer to include support for AUnit [30]. In addition, there are a number of testing techniques which leverage AUnit: *μAlloy* is a mutation testing framework which also provides automated test generation, AlloyFL is a fault localization technique and ARepair is a generate and validate automated repair technique. Outside of AlloyFL, these techniques are deployed as standalone, command-line prototype tools that are run outside of the Analyzer by passing the location of an Alloy model as one of the parameters to the tool. The results from these frameworks are then printed to the command line terminal and sometimes produce artifacts that are saved locally. Therefore, Alloy users do not have access to one centralized integrated development environment (IDE) in which they can actively benefit from these enhancements to test and debug their Alloy models.

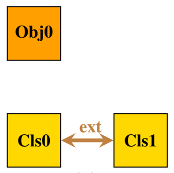
In this paper, we introduce the Analyzer Plus IDE, an integrated development environment for Alloy that combines these AUnit testing frameworks together to form one comprehensive toolset. The benefit of this is two-fold. First, since the Analyzer is the main development environment for Alloy, users will have access to a wide range of debugging options without having to change any behavior related to how they already develop

```
1. sig Class { ext: lone Class }
2. one sig Object extends Class {}
3. pred AllExtObject() {
4. //Each class except Object is a sub-class of Object.
5;   all c: Class - Object | c in c.*ext
6. }
```

(a)

```
1. val Test0 {
2.   some disj Obj0: Object {
3.   some disj Obj0, Cls0, Cls1: Class {
4.     Object = Obj0
5.     Class = Obj0 + Cls0 + Cls1
6.     ext = Cls0->Cls1 + Cls1->Cls0
7.     @cmd:{!AllExtObject[]}
8. }}}
9. @Test Test0: run {Test0}
```

(b)



(c)

Fig. 1. Faulty Java Class Diagram Model and Fault Revealing Test Case

Alloy models. Second, we are able to improve the experience of using these different testing techniques by developing novel interfaces that both contextualize the results and guide the user through the results. Therefore, we can create an Alloy IDE that *aids* in the development of models, rather than one that just *enables* the development of models.

In this paper, we make the following contributions:

- **Integrating AUnit Frameworks**. We incorporate a range of testing frameworks enabled by AUnit into the Analyzer, Alloy's main IDE.
- **Reporting**. We design novel reports that are displayed within the Analyzer to convey the results of different testing frameworks.
- **Case Study**. We explore how the Analyzer Plus IDE can help users find and fix faults in real world faulty models.
- **Open Source**. We release our IDE as an open-source extension to the Analyzer. Analyzer Plus IDE can be found at: https://alloyanalyzerplus.github.io/.

II. Background

In this section, we illustrate key concepts of Alloy, AUnit, and the various testing frameworks that utilize AUnit.

A. Alloy and AUnit

Figure 1 shows a real world faulty Alloy model of a Java class diagram [40]. Signature paragraphs introduce named sets and their relations. Line 1 introduces the named set `Class`, which contains relation `ext` that conveys that each `Class` atom can extend zero or one other `Class` atoms. Line 2 introduces the named singleton (`one`) set `Object` that extends the `Class` signature. Similar to imperative language inheritance, a signature that extends another signature is a subsignature of that signature. Predicates introduce named formulas that can be invoked elsewhere. The predicate `AllExtObject` attempts to state using universal quantification (`all`) that every class except Object extends Object. The fault, in red, uses reflexive transitive closure (`*`) and relational join (`.`) to accidentally convey that every class is a sub-class of itself.

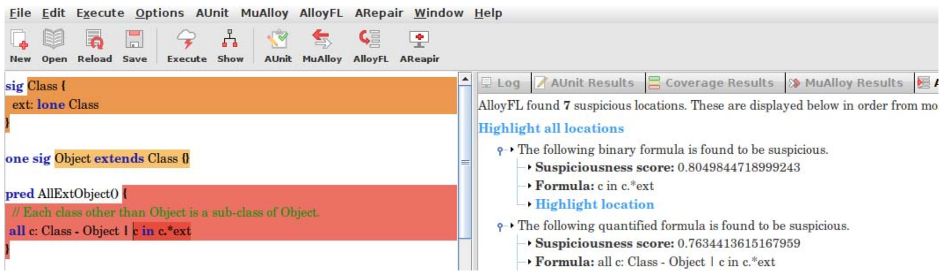
AUnit test cases consists of two components: a valuation, which outlines a specific instance to reason over, and a command, which outlines the formulas under test. A test case passes if the valuation is a valid instance of the command; otherwise, the test fails. In Alloy, there are two types of faults that can appear in a model: (1) *under-constrained* faults in which the model allows instances it should prevent, and (2) *over-constrained* faults in which the model prevents instances it should allow. Therefore, AUnit's format allows for users to directly check for these types of faults without having to (1) enumerate scenarios until finding one that is malformed or (2) enumerate all scenarios and realizing one was missing.

Figure 1 (b) and (c) textually and graphically show a failing test case that highlights the fault in `AllExtObject`. Since both `Cls0` and `Cls1`'s are not connected to `Obj0` through any possible traversal of their `ext` relations, the valuation depicted should not be found as a solution to `AllExtObject`. However, due to the fault, the valuation will incorrectly be found as a valid instance. We have previously introduced native support for AUnit within the Analyzer [30], which expands Alloy's grammar to allow for the declaration of valuation (`val`) paragraphs, support the declaration of test case commands (`@cmd`) within valuations, and to flag Alloy-specific execution commands which are for test cases (`@Test`).

B. AlloyFL

Given a faulty model and a fault revealing AUnit test suite, AlloyFL returns a ranked list of suspicious abstract syntax tree (AST) node locations in the faulty model. To flag locations, AlloyFL supports five different suspiciousness formulas: (1) Tarantula [14], (2) Ochiai [1], (3) Op2 [20], (4) Barinel [2] and (5) DStar [42]. AlloyFL is a hybrid fault localization technique that uses a combination of spectrum-based fault localization and mutation-based fault localization techniques. For spectrum-based fault localization, since Alloy lacks control flow, the suspiciousness score is calculated per predicate paragraph in the model. For mutation-based fault localization, the suspiciousness score is calculated per AST nodes covered by failing tests. For each flagged node, a suspiciousness score is built based on how mutants generated at that location change the test suite's pass/fail behavior. The user can apply a weight to determine how much of the final aggregate score is scaled towards either the spectrum-based or the mutation-based technique. By default, AlloyFL uses the Ochiai suspiciousness formula and a weight of 0.4 (40% mutant-based score and 60% spectrum-based score).

For our faulty class diagram model, AlloyFL returns a ranked list of 7 faulty locations. The most suspicious location is the actual faulty subformula "`c in c.*ext`," with a suspiciousness score of 0.80. This fault can be corrected by either replacing the left operand with "`Object`" or replacing the right operand with "`Object.^ ext`". The next 3 suspicious locations refer to expressions and formulas that relate to "`c in c.*ext`." For example, the second most suspicious location


File Edit Execute Options AUnit MuAlloy AlloyFL ARepair Window Help
New Open Reload Save Execute Show AUnit MuAlloy AlloyFL AReapir
Log AUnit Results Coverage Results MuAlloy Results

```
sig Class {
 ext: lone Class
}

one sig Object extends Class {}

pred AllExtObject() {
 // Each class other than Object is a sub-class of Object.
 all c: Class - Object | c in c.*ext
}
```

```
AlloyFL found 7 suspicious locations. These are displayed below in order from mo
Highlight all locations
  The following binary formula is found to be suspicious.
    Suspiciousness score: 0.8049844718999243
    Formula: c in c.*ext
    Highlight location
  The following quantified formula is found to be suspicious.
    Suspiciousness score: 0.7634413615167959
    Formula: all c: Class - Object | c in c.*ext
```

Fig. 2. AlloyFL Interface for the Faulty Class Diagram Model

is the quantified formula that directly encompasses the faulty formula, which has a score of 0.76.

Motivation for Integration. Unlike the other AUnit extended frameworks, AlloyFL is provided as an extension to the Analyzer and is our motivation for porting the other tools, *μAlloy* and ARepair, into the Analyzer. As a command line tool, AlloyFL could just print a list of locations from most to least suspicious to the screen; however, this would require the user to determine where in the model these locations are. Sometimes an expression or formula could appear multiple times in a model; therefore, tying the command line output to the right portions of the model can take some effort.

In contrast, by integrating AlloyFL into the Analyzer, we can directly highlight the suspicious portion of the model for the user, removing the ambiguity and enhancing the user experience for exploring the ranked list of suspicious locations. To achieve this, we updated the logging interface of the Analyzer to have an "AlloyFL" report tab that displays the ranked list and we automatically highlight locations in the text editor pane based on their suspiciousness score. To illustrate, Figure 2 shows both of these outputs for our example model. A location in the model appears more red the more suspicious the location is found to be. In addition, the user can elect to highlight any single suspicious location.

C. μAlloy

μAlloy generates first-order mutants, generates mutant-killing test cases and performs mutation testing on Alloy models. To generate mutants, *μAlloy* makes changes to the model at the abstract syntax tree (AST) level. For each visited node, *μAlloy* finds all the applicable mutation operators and applies each operator to the node one at a time. The list of currently supported mutant operators can be seen in Table I. For example, if *μAlloy* encounters a unary formula, *μAlloy* would attempt to apply **UOD**, **UOI**, and **UOR** mutant operators to that node. During this process, *μAlloy* automatically discards any mutated model that does not compile. In addition, *μAlloy* also filters all equivalent mutants. Since we are mutating first order logic statements, we can actually use Alloy itself to check if the original formula and the mutated formula are logically equivalent with respect to a user provided scope. If this check fails, Alloy will find a counterexample that highlights the difference between the original model and the mutated model. Through *μAlloy*, users can elect to automatically turn all of these counterexamples into test cases, which guarantees that the user will have a test that can kill all non-equivalent mutants. Once all mutants for a model have been generated, *μAlloy* performs mutation testing using a user provided test suite and reports the mutation score to the user.

For our example model and an test suite of size 8, *μAlloy* generates 18 non-equivalent mutants and prunes 11 equivalent mutants 0.7 seconds. Then, *μAlloy* performs mutating testing over the 8 tests and 18 mutants in 0.4 seconds, which results in a mutation score of 14/18. *μAlloy* generates one additionally test case that kills all 4 remaining non-equivalent mutants. Performing mutation testing on this updated test suite yields a score of 18/18 in 0.45 seconds.

Current Limitations. *μAlloy* is currently supported as a command line tool [37], which is limited to reporting the score, listing the file name of any unkilled mutant and if selected, printing the mutant killing test suite to a file location. However, while this information can be helpful, it is currently missing a lot of context that would enable a user to efficiently apply mutation testing to help improve the quality of their test suite and the accuracy of their model. First, any unkilled mutant is stored as a complete, mutated Alloy model. The user is then left to individually open these mutants and do a differential comparison on their own to figure out the mutated location in the model. Considering that a mutant could be a single character change on one of the lines in the model, having the user take on all the burden to hunt down where the mutated statement is does not scale well, impacting how easily users can rectify any unkilled mutants. Another limitation is that *μAlloy* does not convey which non-equivalent mutants are

TABLE I
MUTATION OPERATORS

Mutation Operator	Description
MOR	Multiplicity Operator Replacement
QOR	Quantifier Operator Replacement
UOR	Unary Operator Replacement
BOR	Binary Operator Replacement
LOR	List Operator Replacement
UOI	Unary Operator Insertion
UOD	Unary Operator Deletion
LOD	Logical Operand Deletion
PBD	Paragraph Body Deletion
BOE	Binary Operand Exchange
IEOE	Imply-Else Operand Exchange

```
5.- all c: Class - Object | c in c.*ext }
5.+ all c: Class - Object | Object in c.*ext }
```

Fig. 3. ARepair Patch for the Faulty Class Diagram Model

connected to their associated mutant killing test case. Instead, all tests appear in a single test suite file and the automatically generated names for tests are in the format "Test[X]."

D. ARepair

ARepair follows the standard generate-and-validate approach, which takes as input a faulty Alloy model and an AUnit test suite with at least one failing test. To fix the model, ARepair uses a greedy, iterative approach where ARepair explores potential patches and applies the first patch that makes some failing tests now pass and no passing test now fail. To generate patches, ARepair first checks if mutants generated by *μAlloy* over just the faulty location can satisfy the greedy choice. If not, ARepair builds an abstract syntax tree representation of the faulty location and creates holes at each level of the AST in a bottom-up fashion. Then, ARepair uses Alloy's grammar and RexGen [39], an Alloy expression generator, to create lists of possible substitutions into each hole. From there, ARepair explores different combinations of substitutions into the holes, searching for a patch that satisfies the greedy choice. If no such patch is found at the current level of the AST, ARepair moves up a level in the AST and repeats the process. In the end, this greedy approach either finds a patch that makes all tests passing or fails to fix the model.

For our faulty class diagram and a test suite with with 14 tests, in the first iteration, ARepair finds that a mutation applied by AlloyFL can make four failing tests now pass but no passing test fail. So, ARepair applies this mutation to the model which mutates "`c in c.*ext`" to "`c != c.*ext`." ARepair then starts its second iteration, in which ARepair does not find a mutant that satisfies the greedy choice. Therefore, ARepair attempts to use the synthesizer to fix the model, which results in "`c != c.*ext`" being replaced with "`Object in c.*ext`." At this point, ARepair determines that all tests have now passed and the model is fixed. Once a patch is found, ARepair then sends the fixed model to the simplifier, which creates a presentable, human readable version of the patch. In our case, the final patch, shown in Figure 3, is semantically equivalent to the correct patch.

Current Limitations. ARepair is currently supported as a command line tool [38]. As ARepair is executing, ARepair displays the intermediate fixes and the final patch to the terminal as completed, updated models. As a result, where the model was changed throughout the process is not readily apparent to the user. By the end, ARepair may have found a successful patch by changing any predicate that is connected to a failing test, changing a signature paragraph, or changing multiple locations throughout the model. All the user knows is that the final iteration displayed passes all tests, assuming ARepair is able to successfully patch the model. Therefore, the burden is entirely on the user to determine how the model

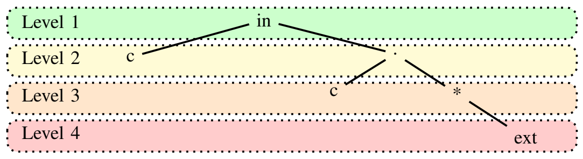


Fig. 4. AST Breakdown of the Faulty Binary Formula Node

was fixed, which the user will likely want to know to help validate the patch. This also hinders the transfer of knowledge to the end user, who would need to work hard to learn what mistake they made in their original faulty model.

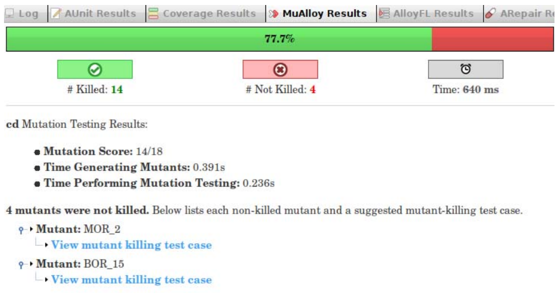
III. Implementation

In this section, we highlight the implementation changes that we have made to support different testing techniques within the Analyzer. These efforts combine together to form our new integrated development environment: Analyzer Plus IDE.

A. AUnit Parser and Abstract Syntax Tree

We built our own parser that enables us to gather additional information valuable for testing that is not available through the original parser of the Alloy model. For AUnit, we intentionally view different first-order logic formulas supported by Alloy in a more nuanced context than Alloy needs to. Specifically, in AUnit, constraints which evaluate to sets are regarded as expressions and constraints which evaluate to true and false are regarded as formulas. We made this distinction because we naturally envisioned different coverage requirements for each: expression coverage criteria relate to the size of sets while formula coverage criteria relate to the truth value. As a result of creating a more fine-grained classification of nodes, we are able to form smaller groups of mutant operators. Since *μAlloy* is integral to both AlloyFL and ARepair, smaller mutant operator groups enables use to apply a more narrow focus during fault localization and repair that can lead to smaller locations and a smaller patches respectively. In addition, these distinctions reduce the number of non-compile-able first order mutants we generate.

To illustrate, Figure 4 shows the AST of the faulty node from our example model, which is the binary formula of the form `[expr] in [expr]`. Our AUnit parser will view "`c in c.*ext`" as a binary *formula*. However, we view the sub-constraint "`c.*ext`" as a binary *expression*. As a result, AUnit will consider these two first-order logic constraints to have different coverage information and to belong to different mutation operator groups. To apply the **BOR** mutation operator on level 1 of the faulty AST (e.g. on formula "`c in c.*ext`"), the set inclusion operator is replaced with the binary set operators set exclusion (`!in`), set equality (`=`) and set inequality (`!=`). Since the right hand and left hand operands for set inclusion are both expressions, we do not additionally create the mutants by substituting in binary logical operators (conjunction (`and`), disjunction (`or`), implication (`=>`) or biconditional (`<=>`)). These logical operators require the left and right hand operands to be formulas to make sense and, in the Analyzer's case, even to compile. To apply the **BOR** mutation operator on level 2 of



77.7%

Killed: 14   # Not Killed: 4   Time: 640 ms

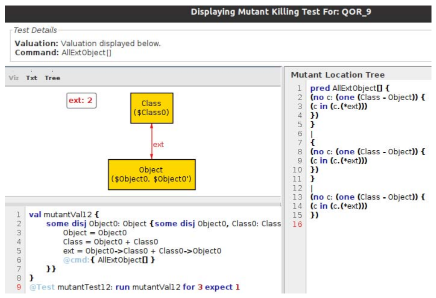
cd Mutation Testing Results:

- **Mutation Score:** 14/18
- **Time Generating Mutants:** 0.391s
- **Time Performing Mutation Testing:** 0.236s

4 mutants were not killed. Below lists each non-killed mutant and a suggested mutant-killing test case.

- **Mutant:** MOR_2
  - View mutant killing test case
- **Mutant:** BOR_15
  - View mutant killing test case

(a) *μAlloy* Execution GUI



Displaying Mutant Killing Test For: QOR_9

Test Details

Valuation: Valuation displayed below.
Command: AllExtObject[]

```
val mutantVal12 {
    some disj Object0: Object {some disj Object0, Class0: Class
        Object = Object0
        Class = Object0 + Class0
        ext = Object0->Class0 + Class0->Object0
        @cmd:{ AllExtObject[] }
    }}
}
@Test mutantTest12: run mutantVal12 for 3 expect 1
```

```
pred AllExtObject[] {
(no c: (one (Class - Object)) {
(c in (c.(*ext)))
})
}
|
{
(no c: (one (Class - Object)) {
(c in (c.(*ext)))
})
}
|
(no c: (one (Class - Object)) {
(c in (c.(*ext)))
})
```

(b) *μAlloy* Test Generation GUI

Fig. 5. *μAlloy* User Interfaces

the faulty AST (e.g. on expression "`c.*ext`"), the relational join operator is replaced with the binary set operators set union (+), set intersection (`&`) and set difference (-) but not with any operator that produces a formula.

However, to the Analyzer, all of these distinctions are irrelevant, as the Analyzer simply needs to store information relevant to translate the model to an equivalent conjunctive normal form formula to hand off to a SAT solver. To help facilitate these testing and debugging operations, the AUnit parser includes support for the Visitor pattern, which allows for the different testing frameworks to define their own actions that occur for each type of node in the AST.

B. Supporting μAlloy

For *μAlloy*, our primary focus is to improve the user's ability to use *μAlloy* for the main benefit mutation testing gives: determining the quality of your test suite, and if needed, helping identify how to improve your test suite.

1) Report Designs: For the *μAlloy* report tab, we first share all the details of the mutation testing execution ranging from the mutation score to the total time taken. Then, the rest of the report tab is dedicated to helping users investigate any unkilled, non-equivalent mutants. Specifically, we present the user with a list of all unkilled mutant and enable the user to drill into each of these by presenting in a separate pop up window that contains (1) the mutant killing test case textually and graphically and (2) the mutated location. As mentioned before, one of the key benefits of performing mutation testing in Alloy compared to imperative languages is that we can use Alloy to not only detect non-equivalent mutants but to also produce a mutant killing test case. However, *μAlloy*'s command line prototype does little to help the user apply this information effectively, mainly by keeping the generated test case and the corresponding mutant separated in outputs presented to the user.

Figure 5 shows examples of the main interfaces for *μAlloy* in Analyzer Plus IDE for our faulty class diagram model. Figure 5 (a) shows the main mutation testing report that appears in the main Analyzer logging pane. For our example, the user is presented with the overall mutation score first, 77.7%. Then, the unkilled mutants are listed. The user can select the "View mutant killing test case" link to investigate each unkilled mutant individually. Selecting this link for our example model and any of unkilled mutant listed, produces the GUI in Figure 5 (b), as all four remaining mutants are killed by this test case.

This pop-up GUI contains all the information the user needs to determine if she would like to add the test case to their model. The automatically generated test case is displayed to the user graphically, for easy inspection, and textually, to easily be copied. In addition, we present a breakdown of the mutant location, so the user can easily identify what part of the model was mutated. Importantly, our interface directly couples the mutant together with the mutant-killing test case. As a result, Analyzer Plus IDE improves the ability of the user to effectively leverage *μAlloy* to strengthen their existing test suite. In addition, since new tests are generated, adding these test to their model can use help reveal potential faults in their model, should a test case unexpectedly fail.

2) Usage: *μAlloy* can be run in two ways. First, the user can press the *μAlloy* icon on the icon menu bar. Second, the user can select the "Execute *μAlloy*" option from the *μAlloy* dropdown menu. By default, *μAlloy* will generate mutants, create mutant-killing test cases and then perform mutation testing using the generated mutants and the original test suite, which does not include any of the test cases generated by *μAlloy*. The only setting the user can configure for *μAlloy* is whether or not to save the test suite to a local file.

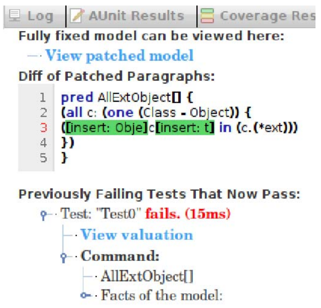
C. Supporting ARepair

For ARepair, our primary focus is to improve the user's ability to easily understand what part of the model was patched, in order for the user to feel comfortable in accepting the automatically repaired model.

1) Report Designs: For the ARepair, we first present how the model is patched to the user, so the user can easily decide if they want to adopt the patch. To achieve this, the user can click the "view patched model" to be presented with a complete model that passes all tests. In addition, to give more context on what changed, we display a "diff" of any patched structures within the Alloy model. In green, we highlight anything that has been inserted into the model. In red, we highlight anything that has been deleted. Second, we want to highlight the previously failing test cases that now pass, as this is the direct behavior we have changed. Figure 6 shows the main report tab for ARepair for our faulty class diagram

TABLE II
PERFORMANCE RESULTS OF AUNIT TESTING FRAMEWORKS FOR CASE STUDY MODELS

Model	#AST	#Flt	#Test	#Fail	Scp	MuAlloy		AlloyFL		ARepair	
						#Mut	Time[s]	Rank	Time[s]	Acc	Time [s]
array	68	1	38	1	3	62	9.9	1:8	4	✓	7.8
cd	52	1	16	5	3	18	0.41	1:7	1.32	✓	1.85
fsm	85	1	21	2	5	72	4.5	1:21	1.2	†	0.6
scl	176	3	66	6	3	91	8.15	1:23	12.9	✓	268.8
sll	40	1	23	4	3	38	1.1	1:9	2.05	✓	0.35



Fully fixed model can be viewed here:
View patched model
Diff of Patched Paragraphs:

```
pred AllExtObject[] {
(all c: (one (Class - Object)) {
([insert: Obje]c[insert: t] in (c.(*ext)))
})
}
```

Previously Failing Tests That Now Pass:
Test: "Test0" fails. (15ms)
View valuation
Command:
AllExtObject[]
Facts of the model:

Fig. 6. ARepair User Interfaces

model. After displaying all patched structures, we then present all previously failing tests that now pass. For each test case, we allow the user to graphically view the valuation of the test case and we additionally present the command of the test case textually. The command is broken up into two components: the command explicitly outlined in the test case and the facts of the model which are always implicitly enforced. In addition, during execution, the main log tab maintains a record of each iteration ARepair undergoes and its associated information, such as the number of failing tests that were updated to passing tests based on the intermediate fix. The information displayed in the log tab, the Analyzer's main execution report log, is the information the command line tool produces.

2) Usage: ARepair can be run in two ways. First, the user can press the ARepair icon on the icon menu bar. Second, the user can select the "Execute ARepair" option from the ARepair drop-down menu. In addition, the user can configure different parameters for the ARepair execution:

- **Search Strategy:** This specifies the type of search the synthesizer should conduct when considering how to combine together substitutions into different holes. The user can toggle between "all-combinations" or "base-choice."
  - For all combinations, ARepair tries all combinations of candidate fragments for all holes until it finds some failing test passed and no passing test failed. All combinations is akin to a brute force approach: it is more likely to find a patch but suffers from runtime scalability issues.
  - For base choice, ARepair holds all holes constant except one hole. For that hole, ARepair explores candidate fragments and picks the one that makes the maximum number of failing tests pass and no passing test fails. Base choice scales better but could miss a patch.
- **Max Try Per Hole**: This parameters is used when the search strategy is "base-choice". The user can specify the maximum number of candidate expressions to consider for each hole during repair as the argument. By default, ARepair uses 1000.
- **Number of Partitions**: This parameter is used when the search strategy is "all-combinations". The user can specify the number of partitions of the search space for a given hole. If By default, ARepair uses 10.
- **Max Try Per Depth**: This parameter is used when the search strategy is "all-combination". The user can specify the maximum number of combinations of candidate expressions to consider for each level/depth of holes during repair. By default, ARepair uses 10000.
- **Save Patch**: The user can specify whether or not to save a copy of the patched model to their local machine. By default, ARepair does save the patch.

IV. CASE STUDY

All models in our case study are real world faulty models created by novice users learning Alloy [40]. One of the most common uses of Alloy is educational. Since Alloy renders its scenarios graphically, the output of the model feels approachable to new users. This allows educators to highlight with graphical illustrations how different formulas work or interact with other formulas. While educational, our models are still reflective of mistakes users make when writing models and based on a recent user study [16], the faults in our study are the same type of fault even expert modelers introduce into Alloy models. For our case study, we focus on two types of faults: (1) models in which the fault is extremely subtle: all models are incorrect due to a single character and (2) a faulty model with multiple mistakes within it, in which we highlight how the frameworks can combine together to help correct it.

Table II highlights the efficacy of the different testing techniques over our running example, the class directory, as well as the illustrative examples in our case study, to give a frame of reference for the performance of these techniques as we step over how a user would realistically leverage them. Column **Model** coveys the model under test. The next 4 columns help convey the size of the model and fault: Column **#AST** is the number of AST nodes the model is comprised of excluding the test suite, **#Flt** is the number of faulty locations, **#Test** is the size of the test suite, which is produced by *μAlloy*, **#Fail** is the number of failing tests and **Scp** is the scope used. The next six columns display performance information for the three frameworks. **#Mut** is the number of non-equivalent mutants generated. **Rank** is a ratio of where the faulty location

```
sig Element {}
one sig Array {
   i2e: Int -> Element, length: Int
}
fact InBound {
   // All indexes should be valid #s.
   all i:Element.~(Array.i2e) {
      i > 0 && i < Array.length
   }
   Array.length >= 0
}
```

(a)

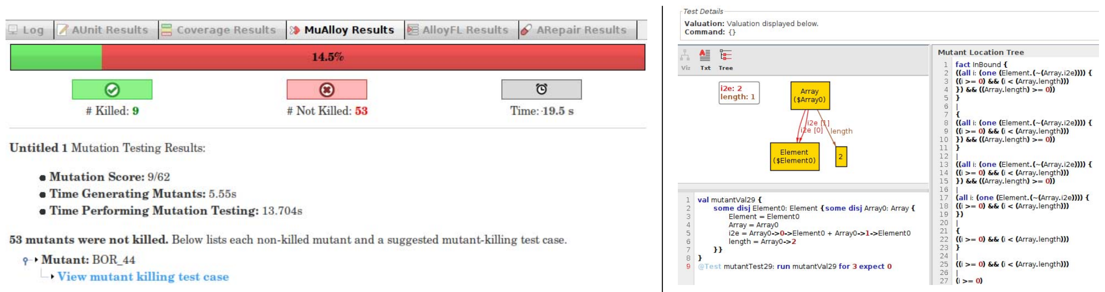
```
one sig FSM {
 start: set State, stop: set State
}
sig State { transition: set State }
fact ValidStartAndStop {
  FSM.start != FSM.stop
  //No transition ends at the start state.
  all s:State | FSM.start != s.transition
  no FSM.stop.transition
}
```

(b)

```
sig List { header: lone Node }
sig Node { link: lone Node }
pred Acyclic(l:List) {
  //Some node terminates the list.
  no l.header or
  some n : Node {
    n in l.header.^link => no n.link
  }
}
```

(c)

Fig. 7. Faulty Models with a Single Character Bug



Untitled 1 Mutation Testing Results:

- **Mutation Score:** 9/62
- **Time Generating Mutants:** 5.55s
- **Time Performing Mutation Testing:** 13.704s

53 mutants were not killed. Below lists each non-killed mutant and a suggested mutant-killing test case.

Mutant: BOR_44
View mutant killing test case

(a) *μAlloy* Results

Test Details
Valuation: Valuation displayed below.
Command: {}

```
val mutantVal29 {
      some disj Element0: Element {some disj Array0: Array {
            Element = Element0
            Array = Array0
            i2e = Array0->0->Element0 + Array0->1->Element0
            length = Array0->2
      }}
}
@Test mutantTest29: run mutantVal29 for 3 expect 0
```

```
fact InBound {
((all i: (one (Element.(~(Array.i2e)))) {
((i >= 0) && (i < (Array.length)))
}) && ((Array.length) >= 0))
}
|
{
((all i: (one (Element.(~(Array.i2e)))) {
((i >= 0) && (i < (Array.length)))
}) && ((Array.length) >= 0))
}
|
((all i: (one (Element.(~(Array.i2e)))) {
((i >= 0) && (i < (Array.length)))
}) && ((Array.length) >= 0))
|
(all i: (one (Element.(~(Array.i2e)))) {
((i >= 0) && (i < (Array.length)))
})
|
{
((i >= 0) && (i < (Array.length)))
}
|
((i >= 0) && (i < (Array.length)))
|
(i >= 0)
```

(b) *μAlloy* Fault Revealing Test

Fig. 8. Debugging a Array Model

is on the ranked list compared to the total number of suspicious locations flagged. **Acc** displays the accuracy of the patch: (✓) means the patch is logically equivalent to the oracle solution, (†) means the patch is plausible, i.e. passes all tests but it not logically equivalent to the oracle patch. **Time[s]** conveys the total execution time, from pressing the button to execute the framework in Analyzer Plus IDE to the time Analyzer Plus IDE finishes populating the report panel, in seconds.

Each framework has a full evaluation done within their respective research papers, that can be referenced for further performance insights across larger scale benchmarks [31, 36, 40]. Rather than focusing on repeating performance evaluations, this section highlights how these different testing interfaces can proactively help users debug real world faulty models.

A. *Single Character Bugs*

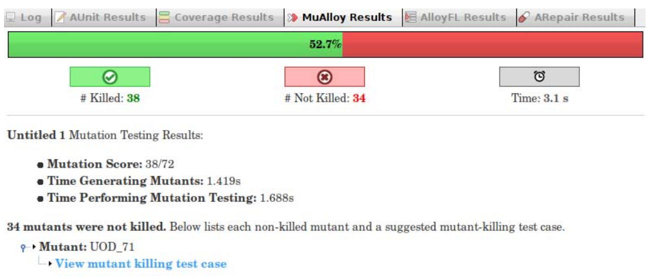
Figure 7 highlights three different models: (a) an array data structure, (b) a finite state machine and (c) a singly linked list data structure. The faulty character, in red, for each model is the result of the user selecting the wrong operator. We include only the faulty predicate or fact in our paper, the complete models can be found on Analyzer Plus IDE's website. In Alloy, a single character change can have a subtle impact on the underlying model. For instance, if the wrong operator makes the model overconstrained, meaning the fault prevents a valid formula from being found, a user may find themselves in the situation where, to notice the fault, the user needs to enumerate all scenarios and realize one or two were missing. Given that predicates often produce hundreds of scenarios, this type of error is extremely difficult to spot in practice.

To highlight how Analyzer Plus IDE can help ease this burden, we illustrate the following steps. First, we draft a single test case for each model. Then, we perform mutation testing using *μAlloy* and save the generated test suite. After providing an oracle for the test suite, we then execute the test suite and investigate any failing test(s) and their corresponding mutant. From there, we step over how investigating the failed test case(s) and applying various AUnit frameworks can lead the user to a correct model.

1) Array: For the array in Figure 7 (b), the user accidentally put greater than instead of greater than or equal (>=). Figure 8 (a) shows the results from running *μAlloy* starting with a single test created that outlines a valid array of size 0. This test case kills just 9 of 62 mutants. To kill the remaining 53 mutants, an additional 33 tests are created. Figure 8 (b) displays the mutant killing test case for the **BOR** mutant which mutates "`i > 0`" to "`i >= 0`." For the displayed test case, the user would label this test as valid, as the indices for the array are assigned values `0` and `1` and the length of the array is `2`. However, when the user runs the corresponding test case, the user will find that it is incorrectly invalid for their model. After labeling all the test cases *μAlloy* produces, the user would discover that the test in Figure 8 (b) is the only failing test.

Since only a single tests fails, the user can jump immediately to applying the mutant to fix their model. If the user goes back to the *μAlloy* results and pulls up the mutant tied to the test in Figure 8 (b), the user will discover that applying the **BOR** mutant "`i >= 0`" will cause their failing test to now pass, fixing the model. Should the user want further confirmation, the user can run AlloyFL, which returns the same location ("`i > 0`") as the most suspicious location, with a score that is notable higher than the next location.

2) FSM: For the finite state machine in Figure 7 (a), the user accidentally put set inequality instead of subset



Killed: 38 # Not Killed: 34 Time: 3.1 s

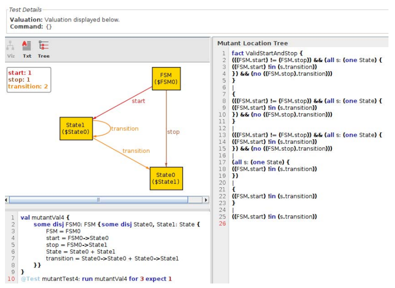
Untitled 1 Mutation Testing Results:

- **Mutation Score:** 38/72
- **Time Generating Mutants:** 1.419s
- **Time Performing Mutation Testing:** 1.688s

34 mutants were not killed. Below lists each non-killed mutant and a suggested mutant-killing test case.

- **Mutant:** UOD_71
  - View mutant killing test case

(a) *µAlloy* Results



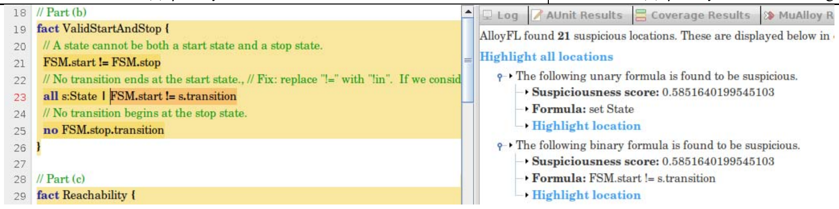
Test Details
Valuation: Valuation displayed below.
Command: {}

```
val mutantVal4 {
    some disj FSM0: FSM {some disj State0, State1: State {
        FSM = FSM0
        start = FSM0->State0
        stop = FSM0->State1
        State = State0 + State1
        transition = State0->State0 + State0->State1
    }}
}
@Test mutantTest4: run mutantVal4 for 3 expect 1
```

Mutant Location Tree

```
fact ValidStartAndStop {
(((FSM.start) != (FSM.stop)) && (all s: (one State) {
((FSM.start) !in (s.transition))
}) && (no ((FSM.stop).transition)))
}
|
{
(((FSM.start) != (FSM.stop)) && (all s: (one State) {
((FSM.start) !in (s.transition))
}) && (no ((FSM.stop).transition)))
}
|
(((FSM.start) != (FSM.stop)) && (all s: (one State) {
((FSM.start) !in (s.transition))
}) && (no ((FSM.stop).transition)))
|
(all s: (one State) {
((FSM.start) !in (s.transition))
})
|
{
((FSM.start) !in (s.transition))
}
|
((FSM.start) !in (s.transition))
```

(b) *µAlloy* Fault Revealing Test



```
// Part (b)
fact ValidStartAndStop {
  // A state cannot be both a start state and a stop state.
  FSM.start != FSM.stop
  // No transition ends at the start state., // Fix: replace "!=" with "!in".  If we consid
  all s:State | FSM.start != s.transition
  // No transition begins at the stop state.
  no FSM.stop.transition
}

// Part (c)
fact Reachability {
```

AlloyFL found **21** suspicious locations. These are displayed below in

Highlight all locations

- The following unary formula is found to be suspicious.
  - **Suspiciousness score:** 0.5851640199545103
  - **Formula:** set State
  - Highlight location
- The following binary formula is found to be suspicious.
  - **Suspiciousness score:** 0.5851640199545103
  - **Formula:** FSM.start != s.transition
  - Highlight location

(c) AlloyFL Results

Fig. 9. Debugging a Faulty Finite State Machine Model

exclusion (`!in`). Figure 9 (a) shows the results from running *µAlloy* with a single test that outlines a valid FSM with two states. This test case kills 38 of 72 mutants. To kill the remaining 34 mutants, an additional 20 tests are created. Figure 9 (b) displays the mutant killing test case for the **BOR** mutant that replaces "`FSM.State != s.transition`" with "`FSM.State !in s.transition`." For this test case, the user would label this test as invalid, as `State1` contains a transition that ends in the start state. However, when the user runs the test case, the user will find that it is incorrectly valid for their model. In fact, after labeling all the test cases *µAlloy* produces, the user will end up with 2 failing tests. The other failing test was generated for the **MOR** mutant that changes the declaration relation from "`transition: set State`" to "`transition: lone State`."

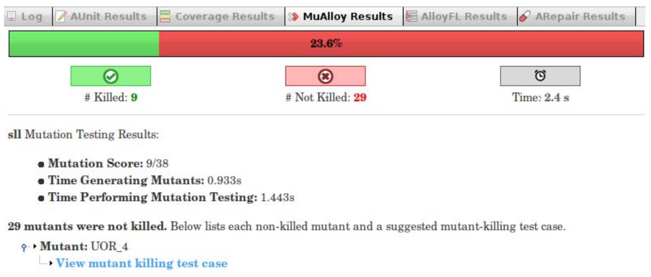
Given that more than one test fails and the failing test cases are tied to two very different mutated locations, the user may jump to running AlloyFL to help narrow in on the faulty behavior. The results of which can be seen in Figure 7 (c). However, the user will discover that two locations are tied for the most suspicious location with a score of 0.59: the location of the **BOR** change and the location of the **MOR** change that produced the two failing test cases. This is where the *µAlloy*'s test generation GUI can help guide the user.

First, to take a closer look at the test case in Figure 9 (b), the user can run the Evaluator, a tool within the Analyzer that returns the concrete value of an expression or formula over a specific instance, on this test case's valuation to reveal that the suspicious formula associated with this test case concretely evaluates to "`State0 != {State0,State1}`." From this, the user can see that the use of set inequality in this formula means that if a state has *multiple* transitions, then that state can incorrectly have the starting state as one of its transitions, as a singleton set will *never* match a set with multiple elements. For the other most suspicious location, the fix implied by the mutant is to limit the multiplicity constraint for the `translation` relation. In this case, the problematic formula "`FSM.State != s.transition`" will not be an issue because the expression "`s.translation`" will either be empty or a singleton set, successfully preventing a state from transitioning to the starting state.

If the user applies either the **BOR** or **MOR** mutant, the user will find that either change will make both failing tests now pass. However, because of the detailed *µAlloy* report, the user can quickly distinguish that the **MOR** patch would be plausible to the user's intentions: applying this mutant would make all tests pass, but limit the functionality of the finite state machine more than the user would want. As a result, the user would adopt the **BOR** mutant change, resulting in a corrected finite state machine model. As seen in Table II, ARepair would actually apply the **MOR** mutant to fix this model, resulting in a plausible patch instead of a correct patch.

3) List: For the singly-linked list in Figure 7 (c), the user accidentally used reflexive transitive closure instead of transitive closure (`^`). Figure 10 (a) shows the results from running *µAlloy* with a single test that outlines a valid list of size 2. This test case kills just 9 of 38 mutants. To kill the remaining 29 mutants, an additional 23 tests are created. Figure 10 (b) displays the mutant killing test case for the **UOR** mutant that mutates "`l.header.*link`" into "`l.header.^link`." For the displayed test case, the user would label this test as invalid, as there is a cycle in the list with the last two nodes pointing back to each other. However, when the user runs the corresponding test case, the user will find that it is incorrectly valid for their model. After labeling all the test cases *µAlloy* produces, the user would discover that there are actually 4 failing test cases. All 4 failing tests contain a list with a cycle and are incorrectly found as valid instances for the faulty model.

Given how many tests fail, the user is likely to use AlloyFL



Killed: 9   # Not Killed: 29   Time: 2.4 s

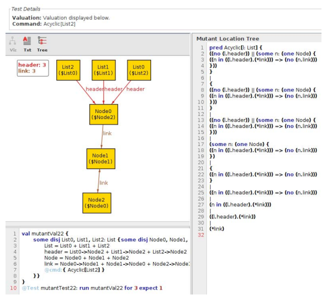
sll Mutation Testing Results:

- **Mutation Score:** 9/38
- **Time Generating Mutants:** 0.933s
- **Time Performing Mutation Testing:** 1.443s

29 mutants were not killed. Below lists each non-killed mutant and a suggested mutant-killing test case.

- **Mutant:** UOR_4
  - View mutant killing test case

(a) *μAlloy* Results



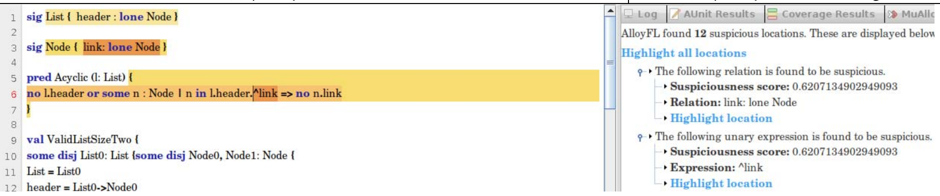
Test Details
Valuation: Valuation displayed below.
Command: Acyclic[List2]

Mutant Location Tree

```
pred Acyclic[l: List] {
((no (l.header)) || (some n: (one Node) {
((n in ((l.header).(*link))) => (no (n.link)))
}))
}
|
{
((no (l.header)) || (some n: (one Node) {
((n in ((l.header).(*link))) => (no (n.link)))
}))
}
|
((no (l.header)) || (some n: (one Node) {
((n in ((l.header).(*link))) => (no (n.link)))
}))
|
(some n: (one Node) {
((n in ((l.header).(*link))) => (no (n.link)))
})
|
{
((n in ((l.header).(*link))) => (no (n.link)))
}
|
((n in ((l.header).(*link))) => (no (n.link)))
|
(n in ((l.header).(*link)))
|
((l.header).(*link))
|
(*link)
```

```
val mutantVal22 {
    some disj List0, List1, List2: List {some disj Node0, Node1,
        List = List0 + List1 + List2
        header = List0->Node2 + List1->Node2 + List2->Node2
        Node = Node0 + Node1 + Node2
        link = Node0->Node1 + Node1->Node0 + Node2->Node1
        @cmd:{ Acyclic[List2] }
    }}
}
@Test mutantTest22: run mutantVal22 for 3 expect 1
```

(b) *μAlloy* Fault Revealing Test



```
sig List {  header : lone Node }

sig Node {  link: lone Node }

pred Acyclic (l: List) {
no l.header or some n : Node | n in l.header.^link => no n.link
}

val ValidListSizeTwo {
some disj List0: List {some disj Node0, Node1: Node {
List = List0
header = List0->Node0
```

AlloyFL found **12** suspicious locations. These are displayed below

Highlight all locations

- The following relation is found to be suspicious.
  - **Suspiciousness score:** 0.6207134902949093
  - **Relation:** link: lone Node
  - Highlight location
- The following unary expression is found to be suspicious.
  - **Suspiciousness score:** 0.6207134902949093
  - **Expression:** ^link
  - Highlight location

(c) AlloyFL Results

Fig. 10. Debugging a Faulty Singly-Linked List Model

```
one sig List { header : lone Node }
sig Node { link: lone Node, elem: one Int }
fact connected  { List.header.*link = Node }
pred Loop(This: List)
    no This.header.link or
    one n: This.header.*link | n.link = n
}
pred Sorted(This: List) {
    all n: This.header.*link {
        one n.link && n.elem <= n.link.elem
}}
pred RepOk(This: List) { Loop[This] and Sorted[This] }
```

Fig. 11. Faulty SLL Model

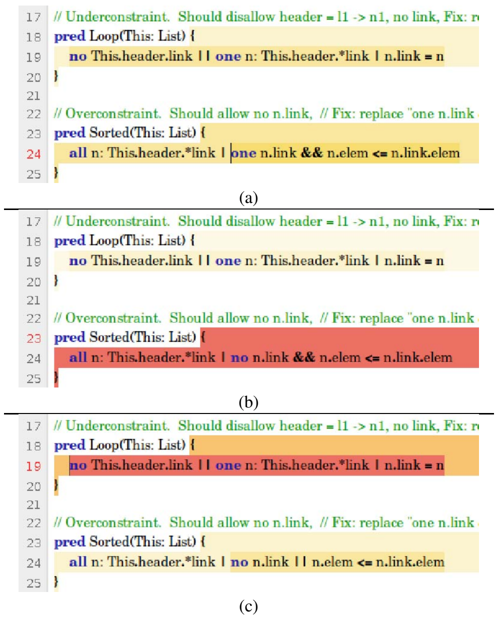
to narrow in on the faulty behavior. Figure 7 (c) displays the results from running AlloyFL on the *μAlloy* test suite. The faulty expression is the tied as the most suspicious location, with a a score of 0.62. The `link` relation declaration is also flagged as the most suspicious location, which helps reiterate that there is an issue relating to how the link's in the list are forming. To choose between which locations to edit, the user can see that the next 3 suspicious locations flagged by AlloyFL all encompass the `^link` expression. If the user checks the *μAlloy* report, the user can then try applying the mutant **UOR**, which the user would narrow in on because it is the only mutant that produces a failing test (Figure 10 (b)) and directly mutates the flagged location.

B. Multiple Faults - SLL

The sorted looped list model (SLL) contains the following predicates: `Loop` which outlines that the last node in the linked list points to itself, `Sorted` which outlines that element should appear in sorted order in the list and `RepOk` which applies both `Loop` and `Sorted` together. Figure 11 depicts a faulty version of this model with 3 different faulty locations across 2 different predicates. To illustrate how Analyzer Plus IDE can help a user triage a model with multiple faulty locations, we first create a single test case that depicts a valid loop list with 3 nodes. Then, we ran *μAlloy* and collected the generated test suite, which produced 65 tests. After labeling these tests as valid or invalid, we end up with 6 failing tests.

Next, we highlight how the AUnit frameworks can be used iterative. First, running AlloyFL on the initial version of the faulty model produces the results in Figure 12 (a). For the most suspcious location, *μAlloy* has a single mutant associated with a failing test for this location; therefore we update "**one** `n.link`" to "**no** `n.link`." After than, running AlloyFL again produces the results in Figure 12 (b). Based on the currently failing tests and the *μAlloy* results, we then changed "&&" to "||". This fully fixes the faulty `Sorted` predicate, but still leaves 5 failing test cases. Running AlloyFL on this new version produces the results in Figure 12 (c). This time, the most suspicious location is connected to multiple failing tests that are derived from different mutants; therefore, there is not a clear mutant to apply. At this point, the user can run ARepair, which produces the results in Figure 13.

Whether a small, subtle bug or a series of larger bugs that noticeably alters the behavior of the model, our case study highlights that Analyzer Plus IDE is able to help guide users towards a corrected model by helping the user: build a strong starting test suite (*μAlloy*), learn more about any faulty behavior based on the difference between their model and the mutant(s) that produce failing test(s) (*μAlloy*), and help the user figure out where to change their model (AlloyFL). Moreover, if the user does not know where to get started to debug their fault, or if the user would rather spend their modeling efforts writing additional predicates, Analyzer Plus IDE can always automatically fix the model for the user (ARepair). The performance of running ARepair on these models can be seen in Table II. For the single character faults, ARepair finds a patch very quickly, as expected since these faults can be repaired by mutation. For the larger SLL model, ARepair patches the model in 4.5 minutes.



```
// Underconstraint.  Should disallow header = l1 -> n1, no link, Fix: r
pred Loop(This: List) {
   no This.header.link || one n: This.header.*link | n.link = n
}

// Overconstraint.  Should allow no n.link,  // Fix: replace "one n.link
pred Sorted(This: List) {
   all n: This.header.*link | one n.link && n.elem <= n.link.elem
}
```

(a)

```
// Underconstraint.  Should disallow header = l1 -> n1, no link, Fix: r
pred Loop(This: List) {
   no This.header.link || one n: This.header.*link | n.link = n
}

// Overconstraint.  Should allow no n.link,  // Fix: replace "one n.link
pred Sorted(This: List) {
   all n: This.header.*link | no n.link && n.elem <= n.link.elem
}
```

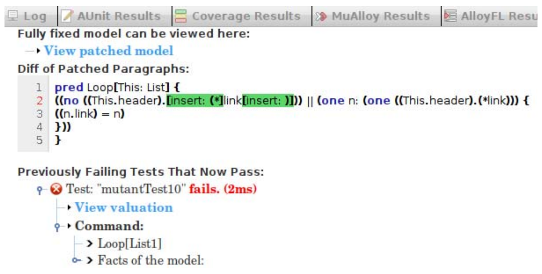
(b)

```
// Underconstraint.  Should disallow header = l1 -> n1, no link, Fix: r
pred Loop(This: List) {
   no This.header.link || one n: This.header.*link | n.link = n
}

// Overconstraint.  Should allow no n.link,  // Fix: replace "one n.link
pred Sorted(This: List) {
   all n: This.header.*link | no n.link || n.elem <= n.link.elem
}
```

(c)

Fig. 12. AlloyFL GUIs for Debugging the SLL Model



Log | AUnit Results | Coverage Results | MuAlloy Results | AlloyFL Resu

Fully fixed model can be viewed here:

View patched model

Diff of Patched Paragraphs:

```
pred Loop[This: List] {
((no ((This.header).[insert: (*]link[insert: )]])) || (one n: (one ((This.header).(*link))) {
((n.link) = n)
}))
}
```

Previously Failing Tests That Now Pass:

Test: "mutantTest10" fails. (2ms)
- View valuation
- Command:
  - Loop[List1]
  - Facts of the model:

Fig. 13. ARepair GUIs for Repairing the SLL Modelt

V. Related Work

In this section, we give an overview of work related to Alloy.

Debugging Techniques for Alloy. The testing frameworks Analyzer Plus IDE integrates into the Analyzer are based on tools that are coupled with AUnit tests. However, there are other debugging techniques for Alloy that use assertions. For repair, ICEBAR extends ARepair to consider built in Alloy assertions over test cases to guide the repair and check candidate patches [10]. ATR is an Alloy repair technique that tries to find patches based on a preset number of templates and uses Alloy assertions as an oracle. BeAFix is an automated repair technique that uses a bounded exhaustive search [11]. TAR is a mutation-oriented repair technique that is aimed at repairing Ally4Fun models, which are educational exercises [7]. FLACK is a fault localization technique that locates faults by using a partial max sat toolset to compare the difference between a satisfying instance of a predicate and a counterexample from an assertion over that predicate [45].

These techniques all focus heavily on using assertions for debugging. Alloy assertions can be a powerful tool for users, but assertions are written in first order logic. Our focus with Analyzer Plus IDE is to improve the native support within the Analyzer for AUnit-based frameworks, as a novice user is more likely to accurately write an AUnit test case than an more robust but complicated assertion. In the future, Analyzer Plus IDE can benefit from incorporating these other frameworks to give users even more ways to test and debug their models.

Scenario Explanation. In a similar vain to testing, there have been a few bodies of work that look to explain why a instance was generated for an Alloy command execution. Analyzer Plus IDE shares a synergy with Amaglam, which aims to help users understand why a instance was found as a solution to an executed command. Specifically for a given instance, Amaglam will generate a trace to outline why a user selected atom or tuple was generated to satisfy the executed command [21]. Recent work also introduced abstract instances for Alloy, which looks to distinguish between portions of a scenario that are present to satisfy the overall, global facts of the model compared to the portions of a instance that are present to satisfy the explicitly executed constraints of the Alloy command [26]. Both these frameworks are complimentary to the overall purpose of Analyzer Plus IDE.

Extensions to the Analyzer for Enumeration. Viewing scenarios is a common "spot check" process for Alloy users, in addition to scenarios being used for test generation and other testing activities. As a result, over the years, there have been several extensions to the Analyzer to influence the way that scenarios are enumerated. A common approach is to try and provide users with an interesting subset of scenarios that may be more valuable to the user: Aluminum enumerates minimum scenarios [22], Hawkeye allows users to influence what instance gets enumerated next based on the current scenario [28], CompoSAT enumerates scenarios with unique coverage [25], and Seabs allows user to enumerate scenarios that differ based on abstract functions [29]. In addition, Reach allows users to enumerate scenarios by size and does not reduce the number of scenarios generated [15].

VI. Conclusion and Future Work

While the Alloy Analyzer is a strength of Alloy compared to other modeling languages, it is far behind the IDEs for imperative code, e.g. Eclipse for Java. As a result, the development process in Alloy can still feel rough and unaccommodating. We believe a key functionality missing from the Alloy Analyzer is a structured format for users to verify the correctness of a model. Therefore, we introduce the Analyzer Plus IDE, an integrated development environment for Alloy models that contains several testing features including test generation, mutation testing, fault localization and repair. Our case study highlights how these testing frameworks can improve developer productivity when debugging models and aid in the creation of more accurate software models. Recently, Alloy was updated to support linear temporal logic. As a result, Alloy is now able to directly express behavioral properties of a system in addition to structural properties of a system. As new versions of AUnit and the AUnit enabled testing frameworks are released, we plan to update Analyzer Plus IDE to build an IDE that also supports testing temporal models.

References

[1] Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C.: A practical evaluation of spectrum-based fault localization. JSS (2009)

[2] Abreu, R., Zoeteweij, P., Van Gemund, A.J.: Spectrum-based multiple fault localization. In: ASE (2009)

[3] Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: 2010 23rd IEEE Computer Security Foundations Symposium. pp. 290–304 (2010)

[4] Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the Android permission system. Formal Asp. Comput. (2018)

[5] Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the android permission system. In: Formal Aspects of Computing. p. 544 (2018)

[6] Bagheri, H., Sullivan, K.: Model-driven synthesis of formally precise, stylized software architectures. Form. Asp. Comput. **28**(3), 441–467 (May 2016)

[7] Cerqueira, J., Cunha, A., Macedo, N.: Timely specification repair for alloy 6. In: Software Engineering and Formal Methods. pp. 288–303 (2022)

[8] Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, Power, ARM, and C++. SIGPLAN Not. **53**(4), 211–225 (2018)

[9] Dini, N., Yelen, C., Alrmaih, Z., Kulkarni, A., Khurshid, S.: In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 1934–1943 (2018)

[10] Gutiérrez Brida, S., Regis, G., Zheng, G., Bagheri, H., Nguyen, T., Aguirre, N., Frias, M.: ICEBAR: Feedback-Driven Iterative Repair of Alloy Specifications. Association for Computing Machinery, New York, NY, USA (2023)

[11] Gutiérrez Brida, S., Regis, G., Zheng, G., Bagheri, H., Nguyen, T., Aguirre, N., Frias, M.: Bounded exhaustive search of alloy specification repairs. In: ICSE. pp. 1135–1147 (2021)

[12] Hao, J., Kang, E., Sun, J., Jackson, D.: Designing minimal effective normative systems with the help of lightweight formal methods. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 50–60. FSE 2016, Association for Computing Machinery, New York, NY, USA (2016)

[13] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2012)

[14] Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: ASE (2005)

[15] Jovanovic, A., Sullivan, A.: Reach: Refining alloy scenarios by size. In: ISSRE (2022)

[16] Mansoor, N., Bagheri, H., Kang, E., Sharif., B.: An empirical study assessing software modeling in alloy. In: International Conference on Formal Methods in Software Engineering. p. To Appear (2023)

[17] Mansoor, N., Saddler, J.A., Silva, B., Bagheri, H., Cohen, M.B., Farritor, S.: Modeling and testing a family of surgical robots: An experience report. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 785–790. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018)

[18] Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class diagrams analysis using Alloy revisited. In: MODELS (2011)

[19] Marinov, D., Khurshid, S.: Testera: a novel framework for automated testing of java programs. In: Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001). pp. 22–31 (2001). https://doi.org/10.1109/ASE.2001.989787

[20] Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. TSE (2011)

[21] Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of "why" and "why not": Enriching scenario exploration with provenance. In: FSE (2017)

[22] Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: ICSE (2013)

[23] Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: LISA (2010)

[24] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Commun. ACM **58**, 66–73 (Mar 2015)

[25] Porncharoenwase, S., Nelson, T., Krishnamurthi, S.: CompoSAT: Specification-guided coverage for model finding. In: FM (2018)

[26] Ringert, J.O., Sullivan, A.: Abstract alloy instances. In: FM. p. To Appear (2023)

[27] Samimi, H., Aung, E.D., Millstein, T.D.: Falling back on executable specifications. In: ECOOP. pp. 552–576 (2010)

[28] Sullivan, A.: Hawkeye: User guided enumeration of scenarios. In: ISSRE (2021)

[29] Sullivan, A., Marinov, D., Khurshid, S.: Solution enumeration abstraction - A modeling idiom to enhance a lightweight formal method. In: ICFEM (2019)

[30] Sullivan, A., Wang, K., Khurshid, S.: AUnit: A Test Automation Tool for Alloy. In: ICST. pp. 398–403 (2018)

[31] Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: ICST. pp. 264–275 (2017)

[32] Sullivan, A., Zaeem, R.N., Khurshid, S., Marinov, D.: Towards a test automation framework for Alloy. In: SPIN. pp. 113–116 (2014)

[33] Taghdiri, M.: Inferring specifications to detect errors in code. p. 144–153. ASE '04, IEEE Computer Society (2004)

[34] Trippel, C., Lustig, D., Martonosi, M.: Security verification via automatic hardware-aware exploit synthesis: The CheckMate approach. IEEE Micro (2019)

[35] Uzuncaova, E., Khurshid, S., Batory, D.: Incremental test generation for software product lines. IEEE Transactions on Software Engineering **36**(3), 309–322 (2010). https://doi.org/10.1109/TSE.2010.30

[36] Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for Alloy. In: ASE. pp. 577–588 (2018)

[37] Wang, K., Sullivan, A., Khurshid, S.: MuAlloy: A Mutation Testing Framework for Alloy. In: ICSE Demo Track. pp. 29–32 (2018)

[38] Wang, K., Sullivan, A., Khurshid, S.: ARepair: A repair framework for Alloy. In: ICSE Demo. pp. 103–106 (2019)

[39] Wang, K., Sullivan, A., Koukoutos, M., Marinov, D., Khurshid, S.: Systematic generation of non-equivalent expressions for relational algebra. In: ABZ. pp. 105–120 (2018)

[40] Wang, K., Sullivan, A., Marinov, D., Khurshid, S.: Fault localization for declarative models in alloy. In: ISSRE. pp. 391–402 (2020)

[41] Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: POPL (2017)

[42] Wong, W.E., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. IEEE Transactions on Reliability (2014)

[43] Zaeem, R.N., Khurshid, S.: Contract-based data structure repair using Alloy. In: ECOOP. pp. 577–598 (2010)

[44] Zave, P.: Using lightweight modeling to understand chord. SIGCOMM Comput. Commun. Rev. **42**, 49–57 (2012)

[45] Zheng, G., Nguyen, T., Gutiérrez Brida, S., Regis, G., Frias, M.F., Aguirre, N., Bagheri, H.: Flack: Counterexample-guided fault localization for alloy models. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 637–648 (2021)